

AFRL-RI-RS-TR-2009-138
Final Technical Report
May 2009



ZERO-KNOWLEDGE PROOF BASED NODE AUTHENTICATION

University of North Texas

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

All *Mathematica* screenshots are from Wolfram *Mathematica* © 2009 Wolfram Research, Inc. www.wolfram.com.
Use of Wolfram *Mathematica* does not imply in any way that Wolfram Research endorses the particular substance or
general quality of this work.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2009-138 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
BRADLY S. PAUL, 2 Lt, USAF
Work Unit Manager

/s/
WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE*Form Approved*
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) MAY 09		2. REPORT TYPE Final		3. DATES COVERED (From - To) Sep 08 – Dec 08	
4. TITLE AND SUBTITLE ZERO-KNOWLEDGE PROOF BASED NODE AUTHENTICATION				5a. CONTRACT NUMBER FA8750-08-1-0260	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 62702F	
6. AUTHOR(S) Eric Ayeh and Kamesh Namuduri				5d. PROJECT NUMBER 4519	
				5e. TASK NUMBER ZK	
				5f. WORK UNIT NUMBER PA	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of North Texas Vice President Research Services Corner of Ave C Chestnut Denton, TX 76203				8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RIGD 525 Brooks Rd. Rome NY 13441-4505				10. SPONSOR/MONITOR'S ACRONYM(S) N/A	
				11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2009-138	
12. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT University of North Texas (UNT) is collaborating with the Air Force Research Laboratory (AFRL) in the design and development of ZKP protocol and in investigating its suitability for airborne networks. During this project, we accomplished the following tasks: (1) Implemented a prototype version of graph isomorphism based ZKP protocol. (2) Analyzed the implementation complexity of the ZKP protocol. (3) Investigated the selection of graphs that are suitable for ZKP implementation. Our experiments related to task (3) indicate that the graphs selected for ZKP implementation must possess specific characteristics. While we developed the basic guidelines for this selection, our results are inconclusive and require additional experiments.					
15. SUBJECT TERMS Airborne Network Protocol, Zero Knowledge Proof, Graph Isomorphism					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 49	19a. NAME OF RESPONSIBLE PERSON Bradly S. Paul, 2Lt, USAF
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) N/A

TABLE OF CONTENTS

1.	SUMMARY	1
2.	INTRODUCTION	2
3.	METHODS, ASSUMPTIONS, AND PROCEDURES	3
4.	RESULTS AND DISCUSSIONS	25
5.	TESTS WITH NAUTY	26
6.	CONCLUSIONS	37
7.	REFERENCES.....	38
8.	APPENDIX A.....	39
9.	APPENDIX B	42
10.	APPENDIX C.....	43
11.	APPENDIX D.....	44

LIST OF FIGURES

Figure 1: Graph G_1 and its adjacency matrix representation	5
Figure 2: Illustration of the case in which $H = G_1 * \rho$ and when the verifier challenges the prover to map H to G_1	6
Figure 3: Graph G_2 in two different forms and its adjacency matrix.....	7
Figure 4: Graph $H = G_1 * \rho$ and its adjacency matrix	8
Figure 5: Illustration of the case in which $H = G_2 * \rho$ and the verifier asks the prover to map H to G_2	8
Figure 6: $H = G_2 * \rho$ and its adjacency matrix representation.....	9
Figure 7: Illustration of the case in which $H = G_1 * \rho$ and the verifier the prover to map H to G_2	9
Figure 8: Illustration of the case in which $H = G_2 * \rho$ and the verifier challenges the prover to map H to G_1	10
Figure 9: Illustration of some of the cases in which the prover provides incorrect answers	11
Figure 10: Computation of σ when $a = b = 0$	13
Figure 11: Computation of σ when $a = b = 1$	14
Figure 12: Computation of σ when $a = 1$ & $b = 0$	14
Figure 13: Computation of σ when $a = 0$ & $b = 1$	15
Figure 14: Adjacency matrix representation of $G_1 = H_1 * \rho^{-1}$	16
Figure 15: Computation of $G_1 = H_1 * \rho^{-1}$	17
Figure 16: Computation of $G_2 = G_1 * \Pi$	18
Figure 17: Computation of $H_2 = G_2 * \rho$	19
Figure 18: Graph G_1 and its adjacency matrix.....	20
Figure 19: Graph H_1 and its adjacency matrix.....	21
Figure 20: Graph G_2 and its adjacency matrix.....	22
Figure 21: Graph H_2 and its adjacency matrix.....	23

1. SUMMARY

A smart way to prove a node's identity without disclosing any information about the secret of that identity is through the use of Zero-Knowledge Proof (ZKP) based authentication protocol. There are several advantages of using this protocol over other authentication schemes. There are also few challenges to overcome in order to make ZKP protocol practical for general use.

During this project, we accomplished the following tasks:

- (1) Implemented a prototype version of graph isomorphism based ZKP protocol.
- (2) Analyzed the implementation complexity of the ZKP protocol.
- (3) Investigated the selection of graphs that are suitable for ZKP implementation.

Our experiments related to task (3) indicate that the graphs selected for ZKP implementation must possess specific characteristics. While we developed the basic guidelines for this selection, our results are inconclusive and require additional experiments.

2. INTRODUCTION

University of North Texas (UNT) is collaborating with the Air Force Research Laboratory (AFRL) in the design and development of ZKP protocol and in investigating its suitability for airborne networks. This research explores the implementation, integration, and certification of ZKP based authentication protocol for airborne networking applications.

The most noteworthy benefit of this collaboration is the *learning opportunities* it is providing to the students at UNT. We hope to continue this collaboration and prepare the students for the challenging jobs in the field of information security at AFRL and other federal organizations.

We have completed Phase I of this project (from 09/16/2008 to 12/31/2008). This report outlines the tasks accomplished during this period.

3. METHODS, ASSUMPTIONS, AND PROCEDURES

We chose to implement ZKP protocol based on Graph Isomorphism (GI). Graph isomorphism has been chosen because of its ease of implementation. The implementation details are presented in section 3.1.

3.1 GRAPH ISOMORPHISM BASED ZKP PROTOCOL IMPLEMENTATION

The Graph isomorphism based ZKP protocol is outlined below.

A graph G_2 is generated from another graph G_1 using a secret permutation named π . In fact, G_2 is obtained by relabeling the vertices of G_1 according to the secret permutation π , while preserving the edges. This pair of graphs (G_1 and G_2) forms the public key pair and the permutation serves as the private key. Another graph named H is either obtained from G_1 or G_2 using another random permutation, say ρ . Once the graph H is obtained, the prover (represents the new node seeking entrance to the network) sends it to the verifier who will challenge him to provide the permutation σ which can map H to either G_1 or G_2 .

For instance, if H is obtained from G_1 and the verifier challenges the prover to map H to G_1 , then $\sigma = \rho^{-1}$. Similarly, if H is obtained from G_2 and the verifier challenges the prover to map H to G_2 , then $\sigma = \rho^{-1}$. On the other hand, if H is obtained from G_1 and the verifier challenges the prover to provide the permutation to map H to G_2 , then $\sigma = \rho^{-1} \circ \pi$, which is a combination of ρ^{-1} and π . In fact, ρ^{-1} will be applied to H to obtain G_1 then, the vertices of G_1 will be modified according to the secret π to get G_2 . Finally, if H is obtained from G_2 and the verifier challenges the prover to map H to G_1 , then $\sigma = \rho^{-1} \circ \pi^{-1}$.

One can notice that in the first two cases, the secret π is not even used. Therefore, a verifier could only be certain of a node's identity after many interactions. Moreover, we can also observe that during the whole interaction process, no clue was given about the secret itself, hence the name Zero Knowledge Proof.

The MATLAB implementation of the ZKP protocol which is similar to the C implementation is presented. The advantage of this implementation is that it allows us to verify the adjacency matrices at each level of the simulation and to evaluate the correctness of the algorithm. Both the C and MATLAB implementations are mainly based on the pseudo-code provided in [1]. In order to illustrate the logic used in the program, we considered a simple graph with 4 nodes. In practice, however, the graphs need to be very large, consisting of several hundreds of nodes, and present a GI complexity in the order of NP-complete. There are three functions that are used in the code, namely, "MakeIsomorphic", "ApplyPermutation", and "CheckGraphs".

a) “MakeIsomorphic” function

This function is used to apply a random permutation (π or ρ) to a specific graph and to obtain another graph that is isomorphic to the original graph. For example, the declaration “ $G_2 = \text{MakeIsomorphic}(G_1, \pi, n)$ ” is used to apply π to the graph G_1 in order to obtain the graph G_2 . Note that the variable ‘ n ’ represents the number of nodes in the graph.

b) “ApplyPermutation” function

This function is used after receiving the challenge response from the prover. In fact, once the verifier receives σ from the prover, his/her objective is to apply that permutation to the graph H to check if he/she will get the expected response (G_1 or G_2). This function is declared as follows, “ $\text{ApplyPermutation}(H, \sigma, n)$ ” where σ is being applied to a graph H .

c) “CheckGraphs” function

After the verifier receives the response σ and applies it to H to obtain another graph that we will name for explanation purpose as R , the “CheckGraphs” function is used to check whether or not $R = G_1$ or $R = G_2$ depending on the case. The function is declared as: “ $\text{CheckGraphs}(G_2, R, n)$.” Here, for example, the function is used to compare both graphs G_2 and R .

The implemented algorithm proceeds as follows: First, the simulator asks for the value of “ n ” which represents the size of the graphs. Once the size is specified, the user has the choice to either start the interaction or to abort the process. The user is then prompted to input the values of the graph G_1 . After all the values are collected, the user is asked for the value of the secret π , then for the value of the random permutation ρ . The user is then asked to choose the graph from which to create the graph H and the graph that the verifier could ask the prover to generate when challenged. After these graphs specified, the user is finally asked for the value of σ . Once the value of σ inserted, the simulator verifies all the inputs and declares if the prover is correct or incorrect. In other words, the simulator verifies if the prover knows the secret or not. Finally, the user is asked to repeat the process if he/she is willing to.

In order to check the correctness of the algorithm we walk through the code with some input examples and validate the results. The simulations were performed with the values of n , π , ρ and the graph G_1 shown below.

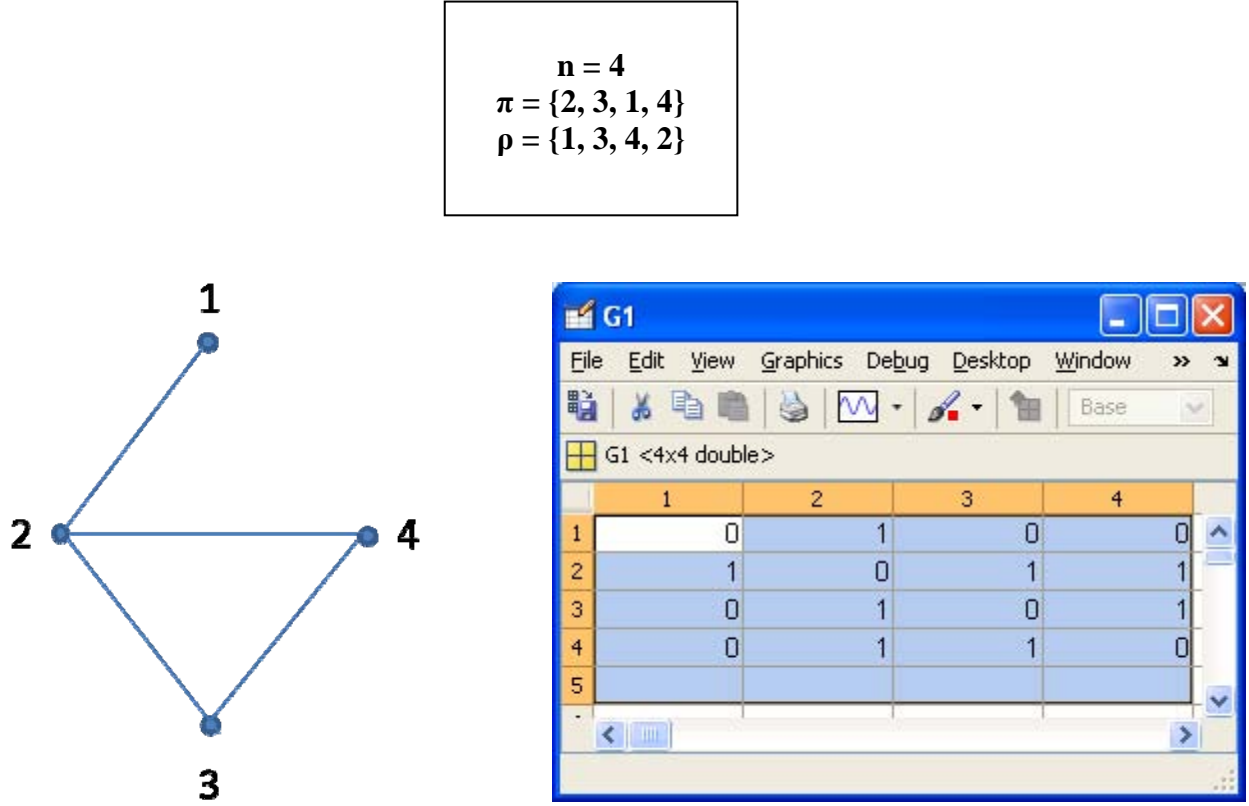


Figure 1: Graph G_1 and its adjacency matrix representation

- If \mathbf{H} is obtained from \mathbf{G}_1 ($\mathbf{H} = \mathbf{G}_1 * \rho$) and the verifier challenges for \mathbf{G}_1 , we will have :
 $\sigma = \rho^{-1} = \{1, 4, 2, 3\}$

To obtain the inverse of $\rho = \{1, 3, 4, 2\}$, we assign indices from 1 to 4 to its vertices, and then process as follows. The first index “1” is at position 1, so its position is not changed. Index 2 is at the position 4; so the second vertex of ρ^{-1} is 4. Index 3 is at the second position; therefore the next vertex in ρ^{-1} is 2. Finally, index 4 is at the third position; therefore, the last element of ρ^{-1} is 3. Following this logic, the vertices of ρ^{-1} are obtained.

- If \mathbf{H} is obtained from \mathbf{G}_2 ($\mathbf{H} = \mathbf{G}_2 * \rho$) and the verifier challenges for \mathbf{G}_2 , we will have
 $\sigma = \rho^{-1} = \{1, 4, 2, 3\}$
- If \mathbf{H} is obtained from \mathbf{G}_1 ($\mathbf{H} = \mathbf{G}_1 * \rho$) and the verifier challenges for \mathbf{G}_2 , we will have
 $\sigma = \rho^{-1} \circ \pi = \{1, 4, 2, 3\} \circ \{2, 3, 1, 4\} = \{2, 4, 3, 1\}$

To calculate σ , we consider the vertices of ρ^{-1} as indices in π and observe which values they are associated with. The process consists of the following steps:

The first vertex of π is 2.
The fourth vertex of π is 4.
The second vertex of π is 3.
The third vertex of π is 1.

- If H is obtained from G_2 ($H = G_2 * \rho$) and the verifier challenges for G_1 , we will have $\sigma = \rho^{-1} \circ \pi^{-1} = \{1, 4, 2, 3\} \circ \{3, 1, 2, 4\} = \{3, 4, 1, 2\}$

Now that all the parameters are set, we will proceed to the actual simulation. Note that the permutations are applied to the rows as well as the columns of the adjacency matrix.

Simulation results will be presented next in order to illustrate the three different cases (i.e. $\sigma = \rho^{-1}$, $\sigma = \rho^{-1} \circ \pi$, and $\sigma = \rho^{-1} \circ \pi^{-1}$).

```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
The value of n is:4
Start the interaction? (1: Yes, 0: No):1
The value of the original graph G1 is: G1=[0 1 0 0;1 0 1 1;0 1 0 1;0 1 1 0]
The value of the secret pi is:[2 3 1 4]
The value of rho is:[1 3 4 2]
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):0
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):0
The value of sigma is:[1 4 2 3]
The prover has proved himself!
Does the verifier want to repeat the process?(1:Yes,0:No):
OVR

```

Figure 2: Illustration of the case in which $H = G_1 * \rho$ and when the verifier challenges the prover to map H to G_1

Once the first simulation is finished, we obtained $\mathbf{G}_2 = \mathbf{G}_1 * \boldsymbol{\pi}$ and $\mathbf{H} = \mathbf{G}_1 * \boldsymbol{\rho}$ as shown in figure 3 and figure 4 respectively. Different scenarios of ZKP protocol are illustrated in figures 5 through 9.

	1	2	3	4
1	0	0	1	1
2	0	0	1	0
3	1	1	0	1
4	1	0	1	0

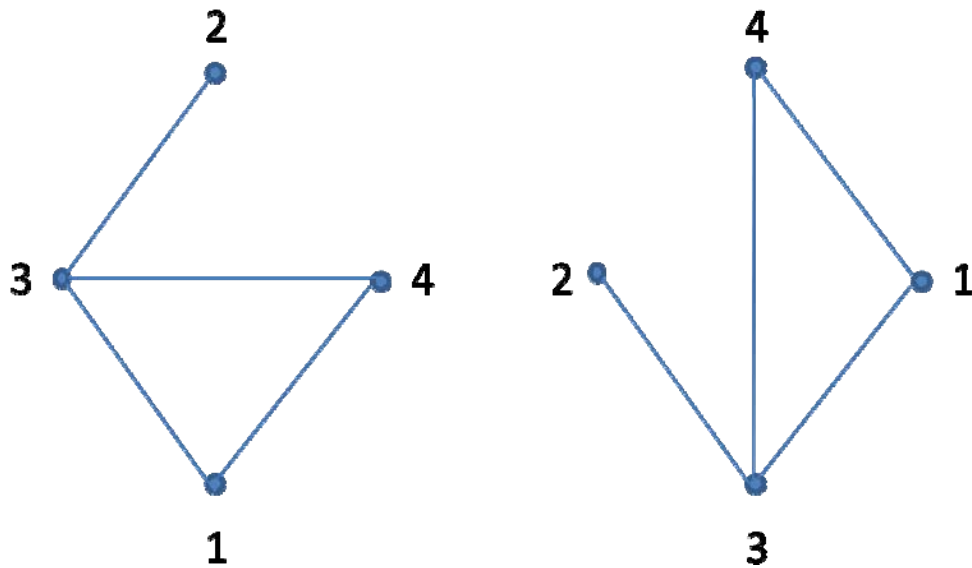
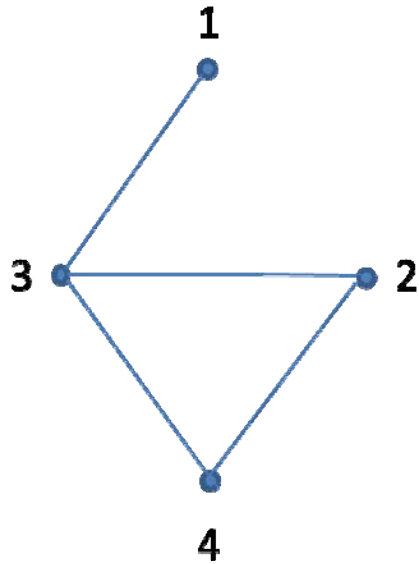


Figure 3: Graph G_2 in two different forms and its adjacency matrix



H <4x4 double>					
	1	2	3	4	
1	0	0	1	0	
2	0	0	1	1	
3	1	1	0	1	
4	0	1	1	0	

Figure 4: Graph $H = G_1 *$ and its adjacency matrix

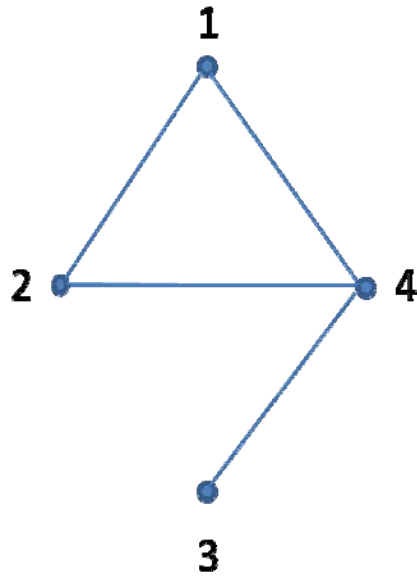
```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

The value of n is:4
Start the interaction? (1: Yes, 0: No):1
The value of the original graph G1 is: G1=[0 1 0 0;1 0 1 1;0 1 0 1;0 1 1 0]
The value of the secret pi is:[2 3 1 4]
The value of ro is:[1 3 4 2]
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):1
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):1
The value of sigma is:[1 4 2 3]
The prover has proved himself!

Does the verifier want to repeat the process?(1:Yes,0:No):
  
```

Figure 5: Illustration of the case in which $H = G_2 *$ and the verifier asks the prover to map H to G_2



	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

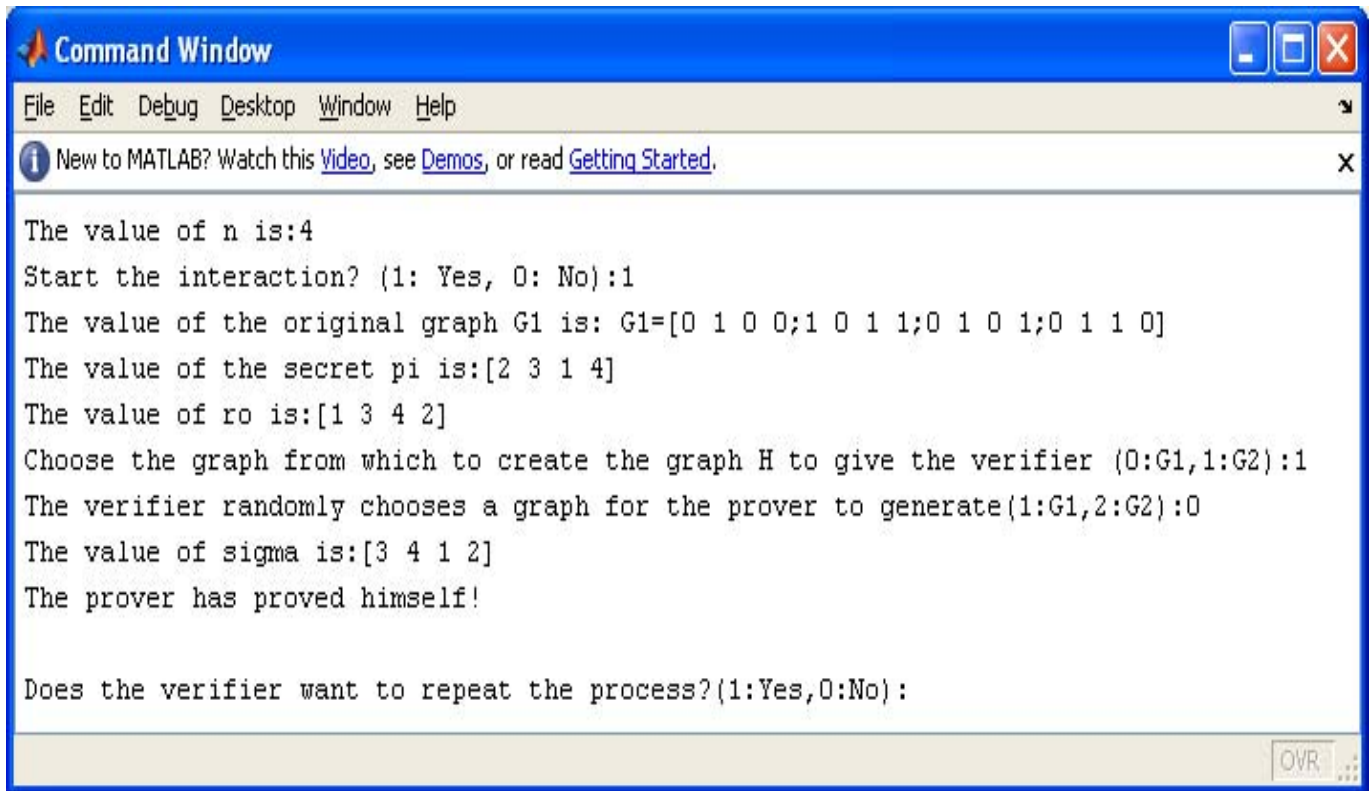
Figure 6: $H = G_2^*$ and its adjacency matrix representation

```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
The value of n is:4
Start the interaction? (1: Yes, 0: No):1
The value of the original graph G1 is: G1=[0 1 0 0;1 0 1 1;0 1 0 1;0 1 1 0]
The value of the secret pi is:[2 3 1 4]
The value of ro is:[1 3 4 2]
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):0
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):1
The value of sigma is:[2 4 3 1]
The prover has proved himself!
Does the verifier want to repeat the process?(1:Yes,0:No):0
>>

```

Figure 7: Illustration of the case in which $H = G_1^*$ and the verifier the prover to map H to G_2



The screenshot shows a MATLAB Command Window with a blue title bar and a menu bar (File, Edit, Debug, Desktop, Window, Help). A message bar at the top says "New to MATLAB? Watch this [Video](#), see [Demos](#), or read [Getting Started](#)." The main area contains the following text:

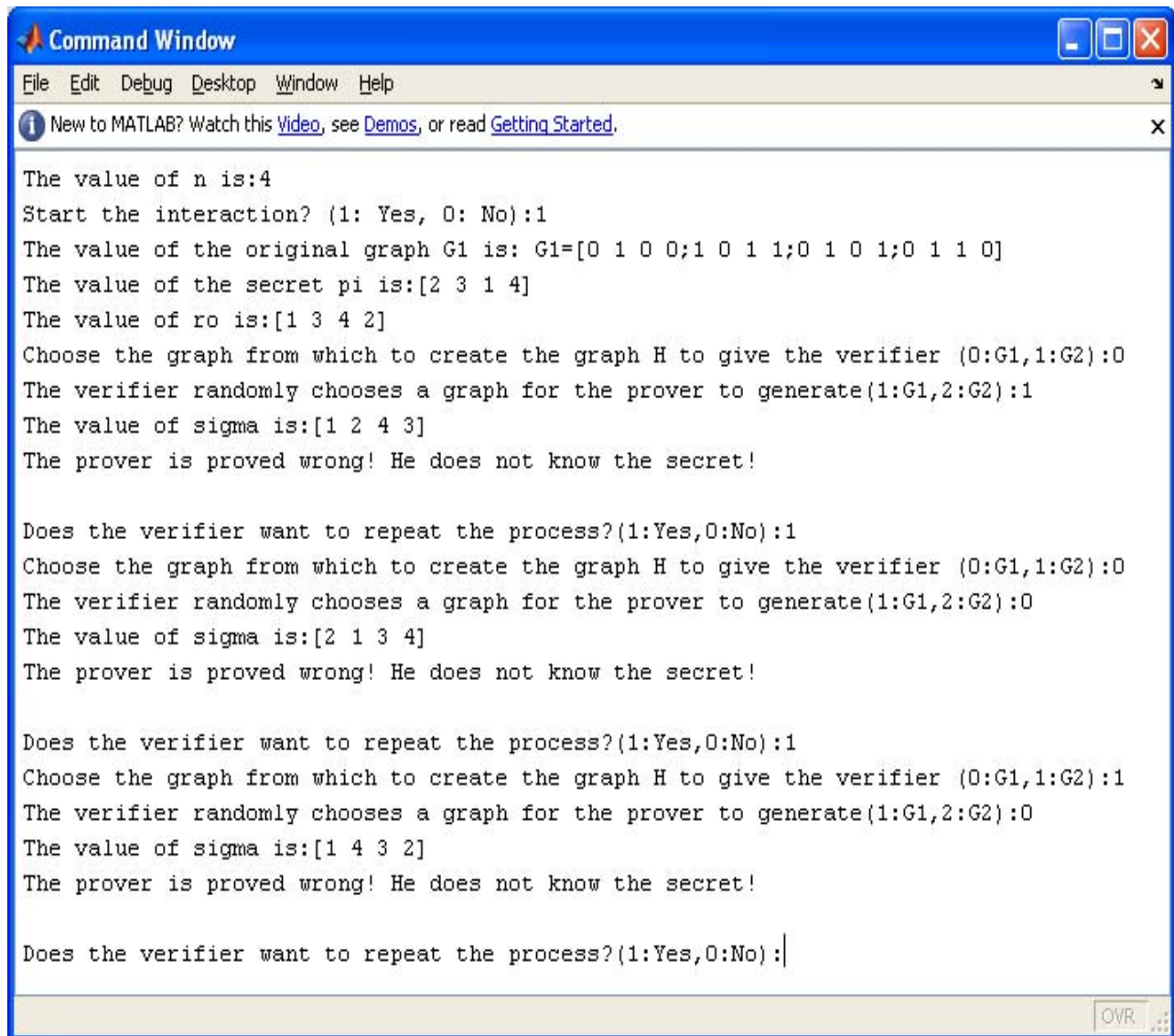
```
The value of n is:4
Start the interaction? (1: Yes, 0: No):1
The value of the original graph G1 is: G1=[0 1 0 0;1 0 1 1;0 1 0 1;0 1 1 0]
The value of the secret pi is:[2 3 1 4]
The value of ro is:[1 3 4 2]
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):1
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):0
The value of sigma is:[3 4 1 2]
The prover has proved himself!

Does the verifier want to repeat the process?(1:Yes,0:No):
```

At the bottom right, there is a button labeled "OVR" with a small icon next to it.

Figure 8: Illustration of the case in which $H = G_2 * \rho$ and the verifier challenges the prover to map H to G_1

Let us now consider some of the cases when the prover provides wrong responses to the challenges.



The screenshot shows a MATLAB Command Window with a blue title bar and a menu bar (File, Edit, Debug, Desktop, Window, Help). A status bar at the bottom right shows 'OVR' and a refresh icon. The main text area contains the following text:

```
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

The value of n is:4
Start the interaction? (1: Yes, 0: No):1
The value of the original graph G1 is: G1=[0 1 0 0;1 0 1 1;0 1 0 1;0 1 1 0]
The value of the secret pi is:[2 3 1 4]
The value of ro is:[1 3 4 2]
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):0
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):1
The value of sigma is:[1 2 4 3]
The prover is proved wrong! He does not know the secret!

Does the verifier want to repeat the process?(1:Yes,0:No):1
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):0
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):0
The value of sigma is:[2 1 3 4]
The prover is proved wrong! He does not know the secret!

Does the verifier want to repeat the process?(1:Yes,0:No):1
Choose the graph from which to create the graph H to give the verifier (0:G1,1:G2):1
The verifier randomly chooses a graph for the prover to generate(1:G1,2:G2):0
The value of sigma is:[1 4 3 2]
The prover is proved wrong! He does not know the secret!

Does the verifier want to repeat the process?(1:Yes,0:No):|
```

Figure 9: Illustration of some of the cases in which the prover provides incorrect answers

One can notice, for example, that when σ is incorrect ($\sigma = \{1, 2, 4, 3\}$) instead of the correct response $\sigma = \rho^{-1} \circ \pi = \{2, 4, 3, 1\}$, the program displays the message asserting that the prover does not know the secret.

Note also that this code has been written for the simple purpose of verifying the steps involved in the ZKP protocol. If we consider the case where a computer acts as the verifier and a user acts as the prover, the user will not be the one inputting the values of π or deciding the graph for the challenge. In fact, such an implementation will only ask the user to input \mathbf{H} , then to input σ .

3.2 CLARIFICATION OF “CREATESIGMA” FUNCTION

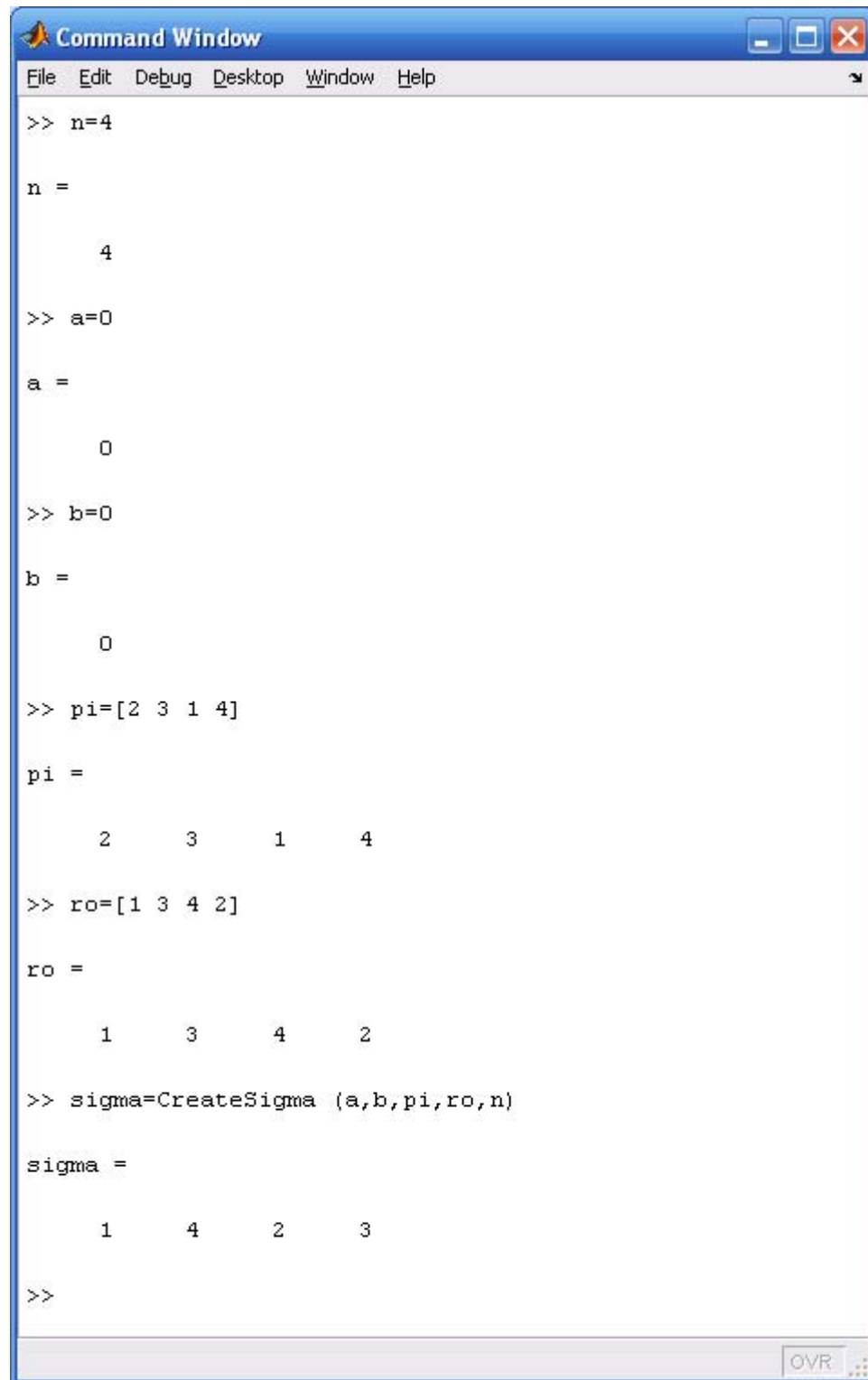
Throughout the implementation of the ZKP protocol, the “CreateSigma” which is used to determine sigma (the response to the verifier’s challenge) is not used. In fact, this function is only utilized for explanation purpose or in order to create sigma easily. Implemented in MATLAB, the “CreateSigma” function has five inputs. The letters “**a**” and “**b**” are used to represent the three different cases that we face in the actual implementation of the protocol, **pi** and **ro** represent the permutations (π and ρ), and finally, “**n**” is the number of nodes in the graphs.

In order to check the correctness of this function, we performed simulations with the values of **n**, π and ρ shown below.

$\begin{aligned} \mathbf{n} &= 4 \\ \pi &= \{2, 3, 1, 4\} \\ \rho &= \{1, 3, 4, 2\} \end{aligned}$
--

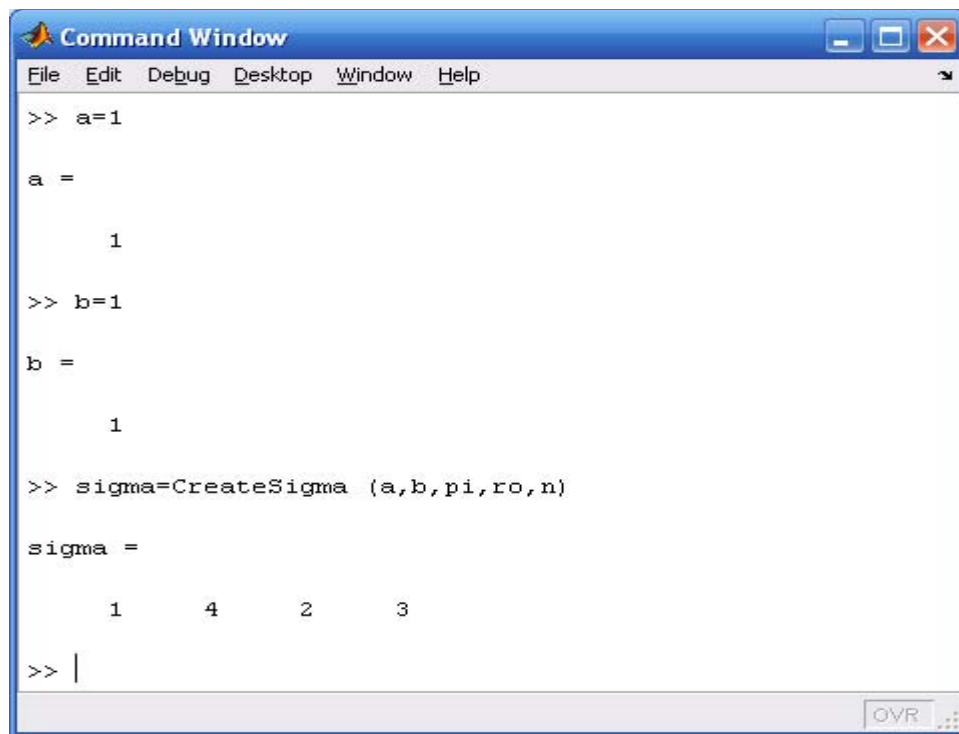
- $\mathbf{a} = \mathbf{b} = 0$ and $\mathbf{a} = \mathbf{b} = 1 \Rightarrow \sigma = \rho^{-1} = \{1, 4, 2, 3\}$
- $\mathbf{a} = 1$ and $\mathbf{b} = 0 \Rightarrow \sigma = \rho^{-1} \circ \pi^{-1} = \{3, 4, 1, 2\}$
- $\mathbf{a} = 0$ and $\mathbf{b} = 1 \Rightarrow \sigma = \rho^{-1} \circ \pi = \{2, 4, 3, 1\}$

The results shown below validate the logic used in the program.



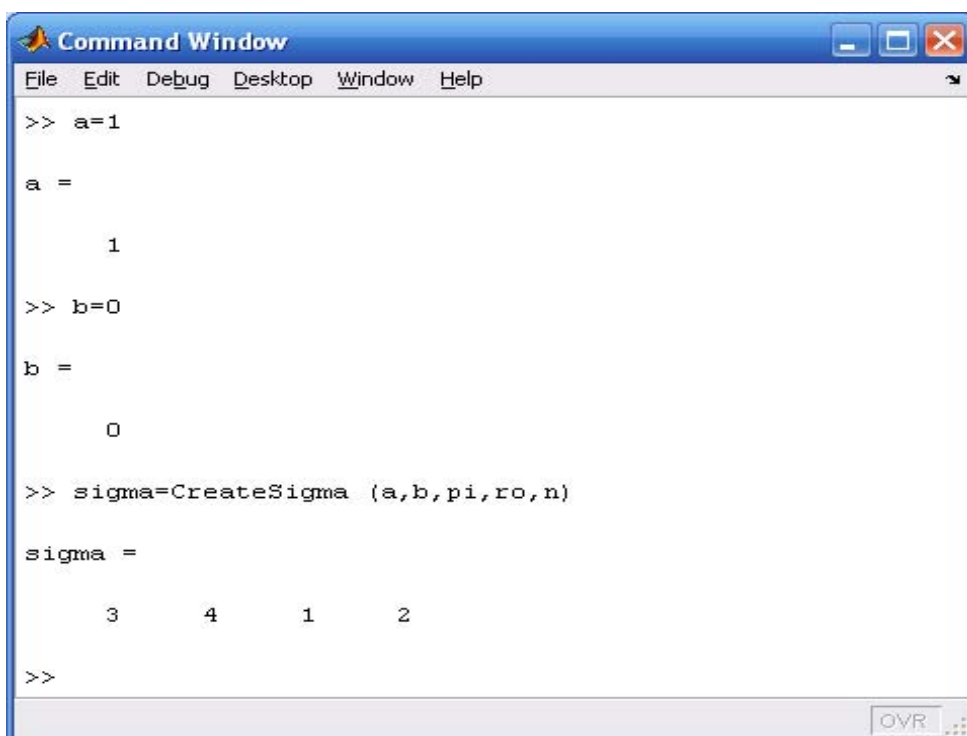
```
Command Window
File Edit Debug Desktop Window Help
>> n=4
n =
    4
>> a=0
a =
    0
>> b=0
b =
    0
>> pi=[2 3 1 4]
pi =
    2    3    1    4
>> ro=[1 3 4 2]
ro =
    1    3    4    2
>> sigma=CreateSigma (a,b,pi,ro,n)
sigma =
    1    4    2    3
>>
```

Figure 10: Computation of σ when $a = b = 0$



```
Command Window
File Edit Debug Desktop Window Help
>> a=1
a =
    1
>> b=1
b =
    1
>> sigma=CreateSigma (a,b,pi,ro,n)
sigma =
    1    4    2    3
>> |
```

Figure 11: Computation of σ when $a = b = 1$



```
Command Window
File Edit Debug Desktop Window Help
>> a=1
a =
    1
>> b=0
b =
    0
>> sigma=CreateSigma (a,b,pi,ro,n)
sigma =
    3    4    1    2
>>
```

Figure 12: Computation of σ when $a = 1$ & $b = 0$

```

>> a=0

a =

     0

>> b=1

b =

     1

>> sigma=CreateSigma (a,b,pi,ro,n)

sigma =

     2     4     3     1

>> |

```

Figure 13: Computation of σ when $a = 0$ & $b = 1$

Applying the “CreateSigma” function to the example given in the reference, the following table is obtained, where the sigma (σ) could only take the values in the colored rows.

Table 1: Updated list of the permutations in the reference

	0	1	2	3	4	5	6	7	8	9
Pi	5	1	9	0	7	4	3	6	2	8
Ro	7	8	1	3	0	5	2	4	6	9
invPi	3	1	8	6	5	0	7	4	9	2
invRo	4	2	6	3	7	5	8	0	1	9
invRo* invPi	5	8	7	6	4	0	9	3	1	2
invRo* Pi	7	9	3	0	6	4	2	5	1	8

3.3 Modifications to the example described in the reference paper [1]

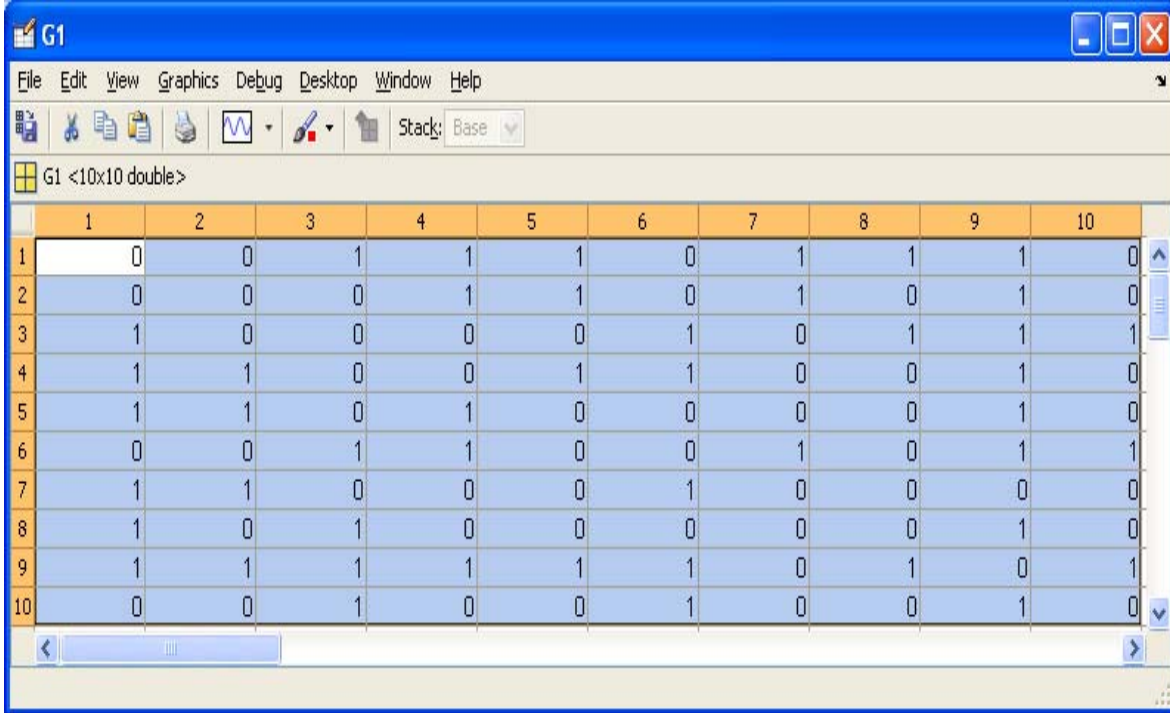
Let us now consider the example given in the reference. Since MATLAB does not support index value of zero, we replace index “0” with “10”. Doing so, we will also be consistent with the adjacency matrix of the graph H_1 shown in Figure 2a (in the reference). Let us assume that this adjacency matrix is obtained by applying the permutation ρ to the original graph G_1 . Therefore, in order to obtain the adjacency matrix of the graph G_1 , we need to compute $H_1 * \rho^{-1}$. From reference [1], we have: $\pi = \{5, 1, 9, 0, 7, 4, 3, 6, 2, 8\}$ and $\rho = \{7, 8, 1, 3, 0, 5, 2, 4, 6, 9\}$

For the MATLAB implementation, we will use: $n = 10, \pi = \{5, 1, 9, 10, 7, 4, 3, 6, 2, 8\}$ and $\rho = \{7, 8, 1, 3, 10, 5, 2, 4, 6, 9\}$

Based on the above assumptions, we have obtained the following permutations using the “CreateSigma” function.

$$\begin{aligned}\rho^{-1} &= \{3, 7, 4, 8, 6, 9, 1, 2, 10, 5\} \\ \rho^{-1} \circ \pi^{-1} &= \{7, 5, 6, 10, 8, 3, 2, 9, 4, 1\} \\ \rho^{-1} \circ \pi &= \{9, 3, 10, 6, 4, 2, 5, 1, 8, 7\}\end{aligned}$$

Applying ρ^{-1} to the adjacency matrix shown in Figure 2a in the reference, we obtain the graph G_1 shown below.



	1	2	3	4	5	6	7	8	9	10
1	0	0	1	1	1	0	1	1	1	0
2	0	0	0	1	1	0	1	0	1	0
3	1	0	0	0	0	1	0	1	1	1
4	1	1	0	0	1	1	0	0	1	0
5	1	1	0	1	0	0	0	0	1	0
6	0	0	1	1	0	0	1	0	1	1
7	1	1	0	0	0	1	0	0	0	0
8	1	0	1	0	0	0	0	0	1	0
9	1	1	1	1	1	1	0	1	0	1
10	0	0	1	0	0	1	0	0	1	0

Figure 14: Adjacency matrix representation of $G_1 = H_1 * \rho^{-1}$

The MATLAB command window displayed while obtaining the original graph G_1 is shown in figure 15.

```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
H1 =
    0    0    0    1    1    1    1    0    1    0
    0    0    0    0    1    0    1    1    0    0
    0    0    0    0    1    1    1    1    0    1
    1    0    0    0    0    1    1    0    0    0
    1    1    1    0    0    1    0    0    1    0
    1    0    1    1    1    0    1    1    1    1
    1    1    1    1    0    1    0    0    0    1
    0    1    1    0    0    1    0    0    0    1
    1    0    0    0    1    1    0    0    0    0
    0    0    1    0    0    1    1    1    0    0

>> invro=[3 7 4 8 6 9 1 2 10 5]

invro =
     3     7     4     8     6     9     1     2    10     5

>> G1=ApplyPermutation (H1,invro,n)

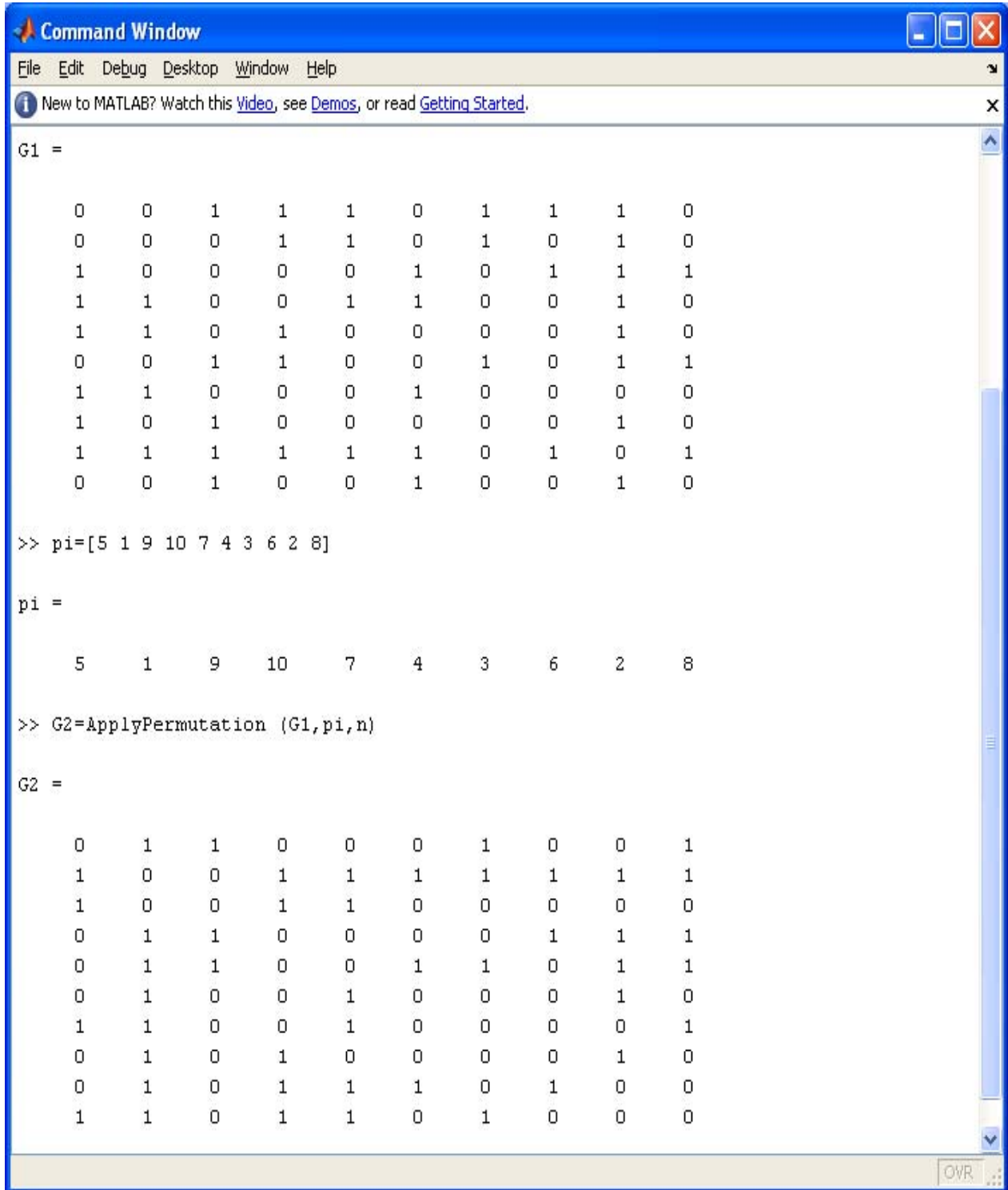
G1 =
    0    0    1    1    1    0    1    1    1    0
    0    0    0    1    1    0    1    0    1    0
    1    0    0    0    0    1    0    1    1    1
    1    1    0    0    1    1    0    0    1    0
    1    1    0    1    0    0    0    0    1    0
    0    0    1    1    0    0    1    0    1    1
    1    1    0    0    0    1    0    0    0    0
    1    0    1    0    0    0    0    0    1    0
    1    1    1    1    1    1    0    1    0    1
    0    0    1    0    0    1    0    0    1    0

>> |

```

Figure 15: Computation of $G_1 = H_1 * \rho^{-1}$

Based on the adjacency matrix of graph G_1 , we will now obtain the adjacency matrices of G_2 , and $H_2 = G_2 * \mathbf{p}$ as shown in figures 16 and 17.



```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

G1 =

    0    0    1    1    1    0    1    1    1    0
    0    0    0    1    1    0    1    0    1    0
    1    0    0    0    0    1    0    1    1    1
    1    1    0    0    1    1    0    0    1    0
    1    1    0    1    0    0    0    0    1    0
    0    0    1    1    0    0    1    0    1    1
    1    1    0    0    0    1    0    0    0    0
    1    0    1    0    0    0    0    0    1    0
    1    1    1    1    1    1    0    1    0    1
    0    0    1    0    0    1    0    0    1    0

>> pi=[5 1 9 10 7 4 3 6 2 8]

pi =

     5     1     9    10     7     4     3     6     2     8

>> G2=ApplyPermutation (G1,pi,n)

G2 =

    0     1     1     0     0     0     1     0     0     1
    1     0     0     1     1     1     1     1     1     1
    1     0     0     1     1     0     0     0     0     0
    0     1     1     0     0     0     0     1     1     1
    0     1     1     0     0     1     1     0     1     1
    0     1     0     0     1     0     0     0     1     0
    1     1     0     0     1     0     0     0     0     1
    0     1     0     1     0     0     0     0     1     0
    0     1     0     1     1     1     0     1     0     0
    1     1     0     1     1     0     1     0     0     0
  
```

Figure 16: Computation of $G_2 = G_1 * \Pi$

```

Command Window
File Edit Debug Desktop Window Help
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

G2 =

    0    1    1    0    0    0    1    0    0    1
    1    0    0    1    1    1    1    1    1    1
    1    0    0    1    1    0    0    0    0    0
    0    1    1    0    0    0    0    1    1    1
    0    1    1    0    0    1    1    0    1    1
    0    1    0    0    1    0    0    0    1    0
    1    1    0    0    1    0    0    0    0    1
    0    1    0    1    0    0    0    0    1    0
    0    1    0    1    1    1    0    1    0    0
    1    1    0    1    1    0    1    0    0    0

>> ro=[7 8 1 3 10 5 2 4 6 9]

ro =

     7     8     1     3    10     5     2     4     6     9

>> H2=ApplyPermutation (G2,ro,n)

H2 =

    0    0    1    0    0    0    1    0    0    1
    0    0    0    0    0    0    1    1    1    1
    1    0    0    1    0    1    0    1    1    0
    0    0    1    0    0    1    0    1    0    0
    0    0    0    0    0    1    0    1    0    1
    0    0    1    1    1    0    0    1    0    1
    1    1    0    0    0    0    0    1    1    0
    0    1    1    1    1    1    1    0    1    1
    0    1    1    0    0    0    1    1    0    1
    1    1    0    0    1    1    0    1    1    0

>>

```

Figure 17: Computation of $H_2 = G_2 * \rho$

Once these adjacency matrices obtained, we used the Wolfram *Mathematica* tool to plot their corresponding graphs as displayed in figures 18 though 21.

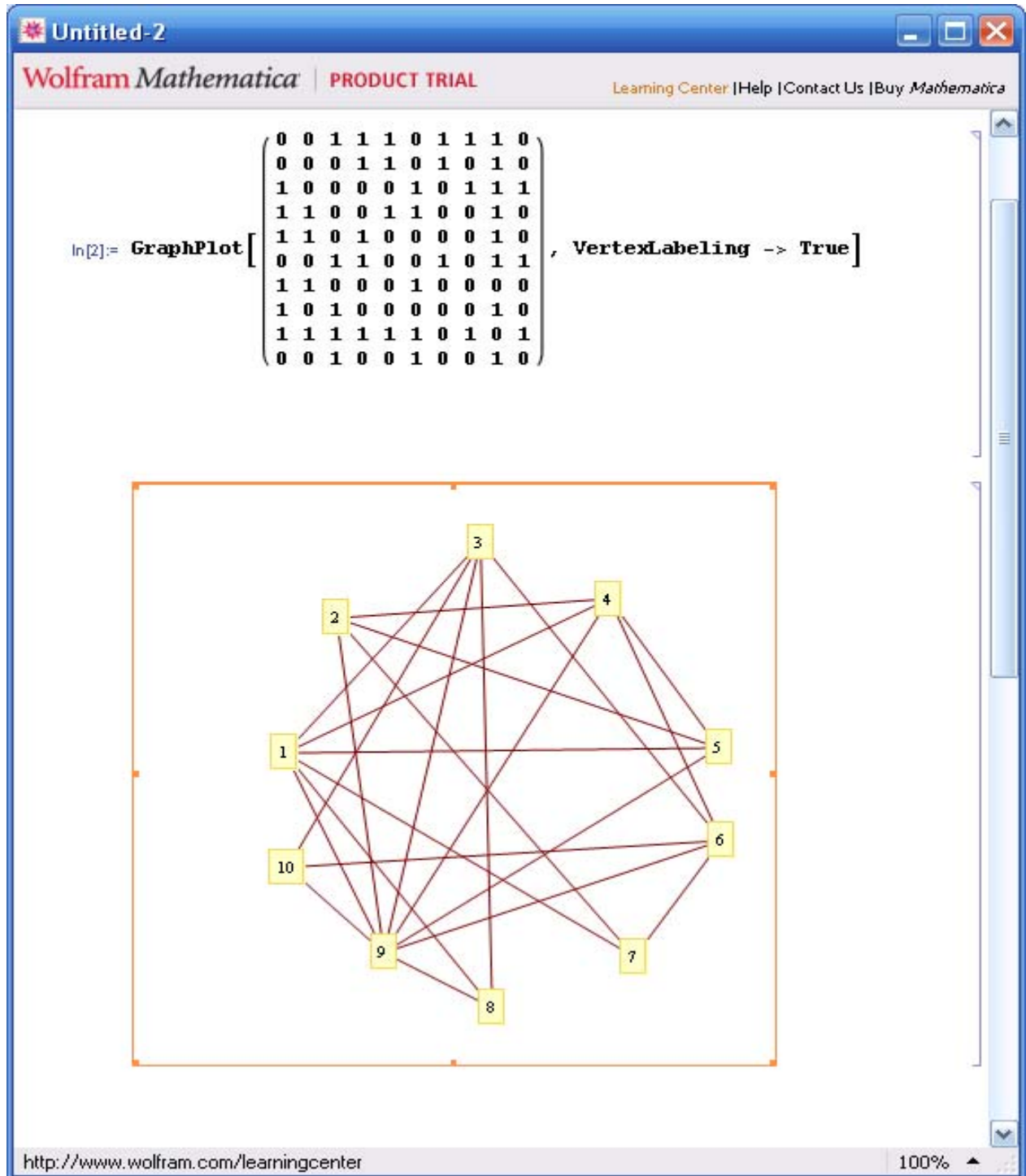


Figure 18: Graph G_1 and its adjacency matrix

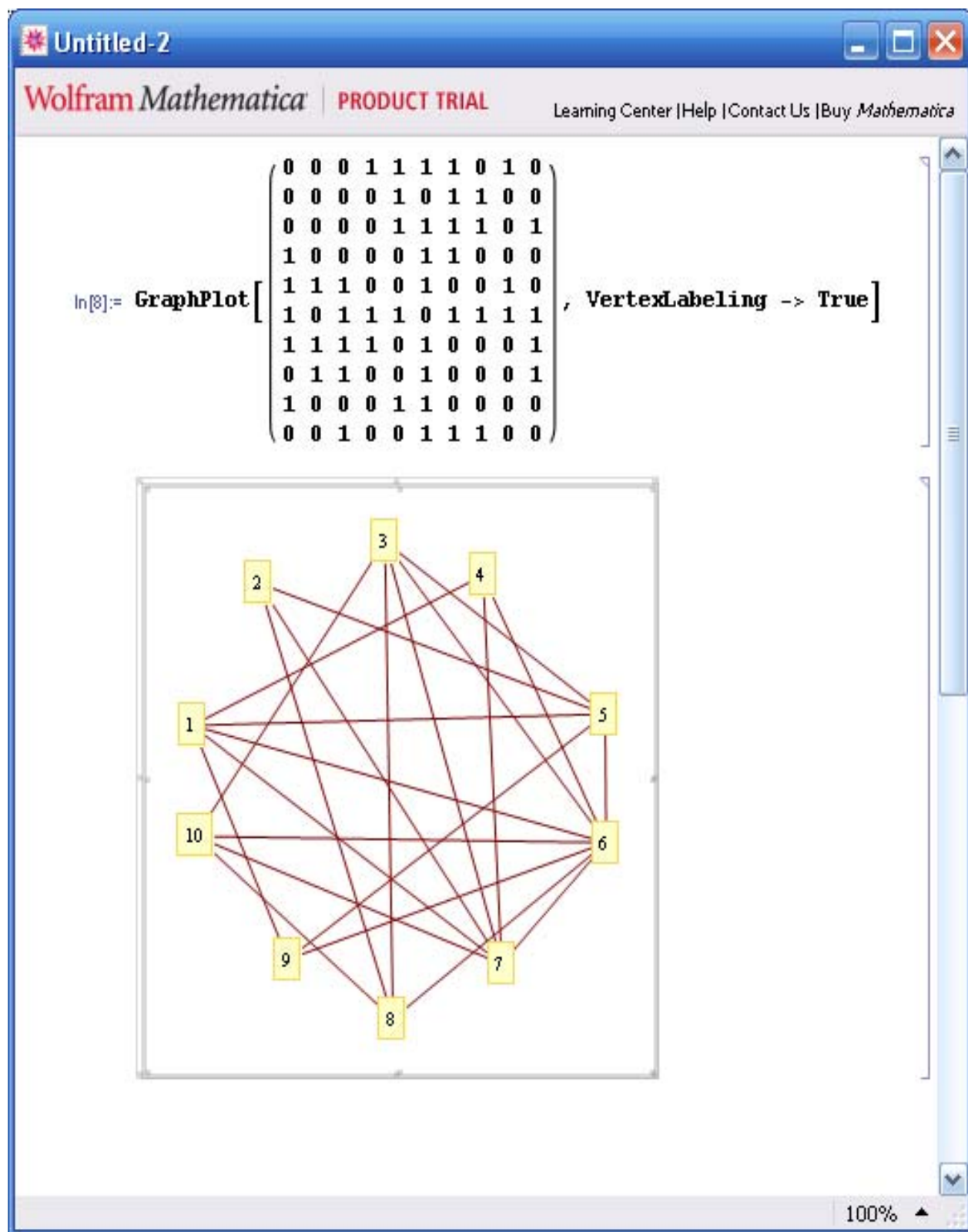


Figure 19: Graph H_1 and its adjacency matrix

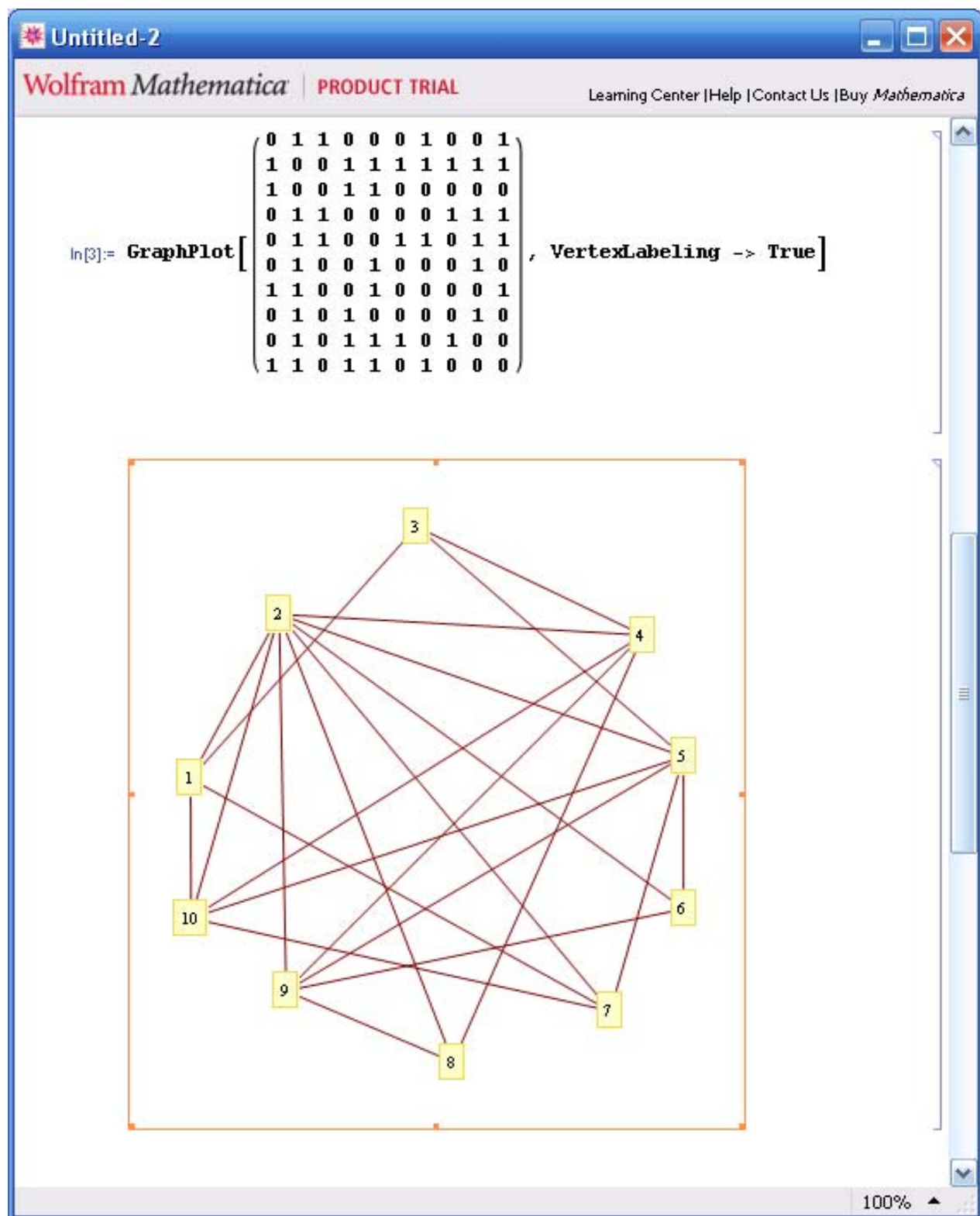


Figure 20: Graph G_2 and its adjacency matrix

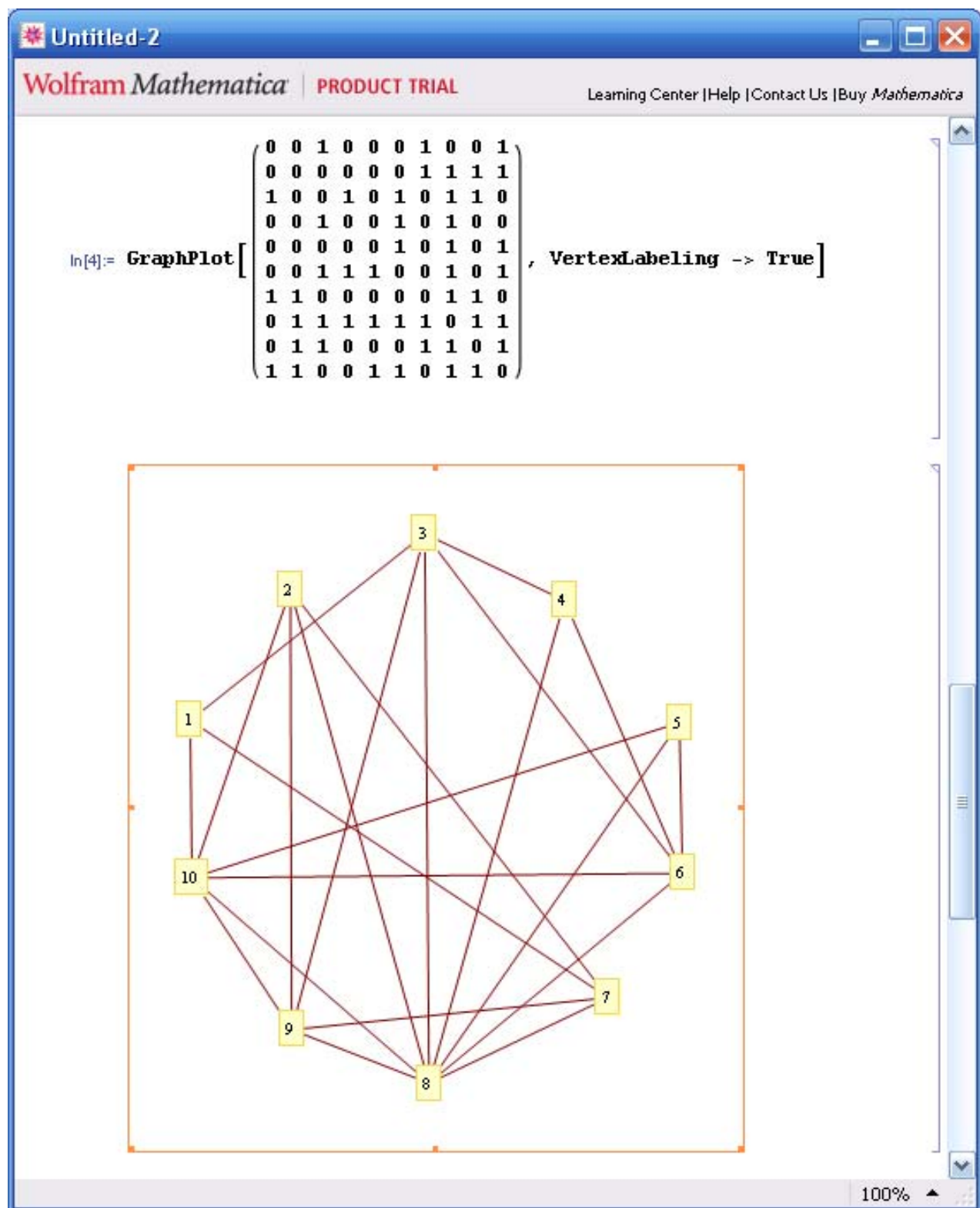


Figure 21: Graph H_2 and its adjacency matrix

The graphs illustrate the characteristics of graph isomorphism. In fact, there is a bijection between the vertices set of \mathbf{G}_1 , \mathbf{G}_2 , \mathbf{H}_1 , and \mathbf{H}_2 . For instance, in order to obtain the graph \mathbf{H}_1 , the vertices of \mathbf{G}_1 were relabeled according to the permutation $\rho = \{7, 8, 1, 3, 10, 5, 2, 4, 6, 9\}$. This means that the node 1 of \mathbf{G}_1 becomes node 7 of the graph \mathbf{H}_1 , node 2 becomes node 8 and so on. This can be seen on the graphs obtained above. However, since we have rearranged the nodes so as to have them in order (i.e. from 1 to 10), in order to verify the isomorphism, one should consider the edges of each of the nodes. For example node 1 of \mathbf{G}_1 has four edges as node 7 of \mathbf{H}_1 . Moreover if you consider node 9 of \mathbf{G}_1 which has the highest number of edges (8), it becomes node 6 of \mathbf{H}_1 .

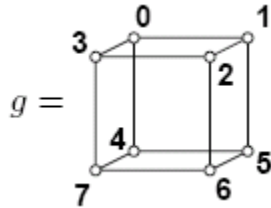
4. RESULTS AND DISCUSSIONS

We have successfully implemented using the MATLAB application the ZKP protocol. The developed code could be modified in order to obtain a better user friendly interactive system. However, the graph isomorphism scheme that we have used as the basis for the implementation of the ZKP protocol, requires appropriate selection of graphs.

Our goal is to find out the types of graphs whose isomorphism is hard to break using tools such as Nauty [2]. We first started with regular graphs that have low automorphs, as suggested by Michael Gudaitis at AFRL. Regular graphs having only one automorph (i.e. only isomorphic to itself), are one type of graphs that we studied, since they have higher graph isomorphism complexity. However, algorithms such as Nauty exploit the patterns in graphs to break the graph isomorphism.

5. TESTS WITH NAUTY

Prior to testing the performance of Nauty using regular graphs with only one automorph, we will go through one of the examples provided in the *Nauty* User's Guide (Version2.2) by Brendan D. McKay in order to have an understanding of how the program is run.



Let g a graph as represented in the above figure. Our goal is to find all the automorphs to g using the Nauty program. The program provides different commands that are used in order to run a test. Obtained from the *Nauty* User's Guide, below are the lists and brief explanations of the commands that we have used in our tests.

n= #: Set value of n (number of nodes). The maximum value is installation-defined.

g: Read the graph g .

c,-c: Set option *getcanon* to TRUE or FALSE, respectively. This tells Nauty whether to find a canonical labeling or just the automorphism group. The default is FALSE.

a,-a: Set option *writeautoms* to TRUE or FALSE, respectively. This tells Nauty whether to display the automorphisms it finds. The default is TRUE.

m,-m: Set option *writemarkers* to TRUE or FALSE, respectively. This tells Nauty whether to display the level markers "level . . ."

p,-p: Set option *cartesian* to TRUE or FALSE, respectively. This tells Nauty to use the "cartesian" form when writing the automorphisms.

x: Execute Nauty. Depending on the values of the *writeautoms* and *writemarkers* options, the automorphism group will be displayed while Nauty is running. When Nauty returns, **dreadnaut** will display some statistics about it. Depending on your system, the execution time is also displayed.

@: Copy h , if defined, to $h0$. See the description of the **#** command for more information.

#: Compare the labeled graphs h and $h0$. Both must have been already defined (using **x** and **@**).

The complete process for testing two graphs $g1$ and $g2$ for isomorphism is this:

```
enter g1;  
c x @ (select getcanon option, execute Nauty, copy  $h$  to  $h0$ );  
enter g2;  
x # (execute Nauty, compare  $h$  to  $h0$ ).
```

##: This is the same as **#** except that, if h is identical to $h0$, you will also be given an isomorphism from $g1$ to $g2$. This is in the form of a sequence of pairs' v_i-w_i , where v_i is a vertex of $g1$ and w_i is a vertex of $g2$. The vertex-numbering origin in force when $h0$ was created is used for $g1$, whilst the origin now in force is used for $g2$.

The outputs obtained when the program is run can be interpreted as follows:

int numorbits: The number of orbits of the automorphism group.

int numgenerators: The number of generators found.

long numnodes: The total number of tree nodes generated.

long numbandleaves: The number of leaves of the tree which were generated but were useless in the sense that no automorphism was thereby discovered and the current-best-guess at the canonical labeling was not updated.

int maxlevel: The maximum level of any generated tree node. The root of the tree is on level one.

long tctotal: The total size of all the target cells in the search tree. The difference between this value and *numnodes* provides an estimate of the efficiency of Nauty's search-tree pruning.

long canupdates: The number of times the program's idea of the "best candidate for canonical label" was updated, including the original one.

Test1: (Provided in the *Nauty* User's Guide)

```
eestudent@eeuser-LINUX:~/Desktop/nauty22$ ./dreadnaut
Dreadnaut version 2.2 (32 bits).
> n=8 g
  0 : 1 3 4;
  1 : 2 5;
  2 : 3 6;
  3 : 7;
  4 : 5 7;
  5 : 6;
  6 : 7.
> p a c m
> x
  0 1 5 4 3 2 6 7
level 3:  6 orbits; 3 fixed; index 2
  0 3 2 1 4 7 6 5
level 2:  4 orbits; 1 fixed; index 3
  1 0 3 2 5 4 7 6
level 1:  1 orbit; 0 fixed; index 8
1 orbit; grpsize=48; 3 gens; 10 nodes; maxlev=4
tctotal=20; canupdates=1; cpu time = 0.00 seconds
```

The results of this test show that there are three isomorphisms to the graph **g**. In fact, if we consider the graph **g** we can clearly notice that switching 2-5 and 3-4 does not affect the structure of the graph.

Now that we have an idea of how the program works, let begin testing with the regular graphs with one automorph. We use the notation $N_{xx}E_{yy}$ to represent a regular graph, where xx denotes the number of nodes and yy denotes the number of edges for each node.

Test2: Graph-N10E5

```
eestudent@eeuser-LINUX:~/Desktop/nauty22$ ./dreadnaut
Dreadnaut version 2.2 (32 bits).
> n=10 g
 0 : 1 2 3 5 7;
 1 : 3 4 6 8;
 2 : 5 6 7 9;
 3 : 4 5 6;
 4 : 7 8 9;
 5 : 8 9;
 6 : 8 9;
 7 : 8 9.
> p a m c
> x
level 1: 1 cell; 10 orbits; 0 fixed; index 1/10
10 orbits; grpsize=1; 0 gens; 11 nodes (7 bad leaves); maxlev=2
tctotal=10; canupdates=3; cpu time = 0.00 seconds
```

From the interpretation of the outputs, we can say that this graph does not have any other automorphisms other than itself.

Test3: Graph-N11E6

```
> n=11 g
 0 : 2 3 4 7 8 9;
 1 : 3 4 5 8 9 10;
 2 : 4 6 8 9 10;
 3 : 5 7 9 10;
 4 : 6 7 8;
 5 : 6 7 9 10;
 6 : 8 10;
 7 : 9;
 8 : 10.
> p
> a
> c
> m
> x
11 orbits; grpsize=1; 0 gens; 1 node; maxlev=1
tctotal=0; canupdates=1; cpu time = 0.00 seconds
```

Test4: Graph-N12E6

```
> n=12 g
0 : 1 2 3 5 7 10
0 : 1 2 3 5 7 10;
1 : 3 4 6 8 11;
2 : 5 6 7 9 10;
3 : 4 5 6 11;
4 : 7 8 9 10;
5 : 8 9 10;
6 : 8 9 11;
7 : 8 9 10;
8 : 11;
9 : 11;
10 : 11.
> p a c m
> x
level 1: 1 cell; 12 orbits; 0 fixed; index 1/12
12 orbits; grpsize=1; 0 gens; 13 nodes (9 bad leaves); maxlev=2
tctotal=12; canupdates=3; cpu time = 0.00 seconds
```

Test5: Graph-N13E6

```
> n=13 g
0 : 2 4 6 7 10 11;
1 : 3 6 7 8 9 12;
2 : 4 7 8 9 11;
3 : 5 6 9 10 12;
4 : 6 8 10 11;
5 : 7 8 9 11 12;
6 : 8 12;
7 : 9 10;
8 : 10;
9 : 11;
10 : 12;
11 : 12.
> p a c m
> x
level 1: 1 cell; 13 orbits; 0 fixed; index 1/13
13 orbits; grpsize=1; 0 gens; 14 nodes (11 bad leaves); maxlev=2
tctotal=13; canupdates=2; cpu time = 0.00 seconds
```

Test6: Graph-N14E7

```
> n=14 g
 0 : 1 2 3 5 7 10 12;
 1 : 3 4 6 8 11 12;
 2 : 5 6 7 9 10 13;
 3 : 4 5 6 11 12;
 4 : 7 8 9 10 13;
 5 : 8 9 10 12;
 6 : 8 9 11 13;
 7 : 8 9 10 12;
 8 : 11 12;
 9 : 11 13;
10 : 11 13;
11 : 13;
12 : 13.
> p a c m
> x
level 1:  1 cell; 14 orbits; 0 fixed; index 1/14
14 orbits; grpsize=1; 0 gens; 15 nodes (12 bad leaves); maxlev=2
tctotal=14; canupdates=2; cpu time = 0.00 seconds
```

As per the results of the different tests, these graphs exhibit the property of having only one automorph. However, these results were obtained by Nauty in less than a second. We have then decided using the same graphs to test how long it will take the program to check if two graphs are isomorphic. The tests for finding graph isomorphism have only been performed on N10E5 and N14E7.

Test7: Graph-N10E5

Using MATLAB, a random permutation ({1 0 2 3 4 5 8 7 9 6}) has been chosen and applied to graph N10E5 in order to obtain another graph N10E5'.

```
n =
```

```
10
```

```
>> GB=[0 1 1 1 0 1 0 1 0 0;1 0 0 1 1 0 1 0 1 0;1 0 0 0 0 1 1 1 1 0
1;1 1 0 0 1 1 1 0 0 0;0 1 0 1 0 0 0 1 1 1;1 0 1 1 0 0 0 0 1 1;0 1
1 1 0 0 0 0 1 1;1 0 1 0 1 0 0 0 1 1;0 1 0 0 1 1 1 1 0 0;0 0 1 0 1
1 1 1 0 0]
```

```
GB =
```

0	1	1	1	0	1	0	1	0	0
1	0	0	1	1	0	1	0	1	0
1	0	0	0	0	1	1	1	0	1
1	1	0	0	1	1	1	0	0	0
0	1	0	1	0	0	0	1	1	1
1	0	1	1	0	0	0	0	1	1
0	1	1	1	0	0	0	0	1	1
1	0	1	0	1	0	0	0	1	1
0	1	0	0	1	1	1	1	0	0
0	0	1	0	1	1	1	1	0	0

```
>> sigma=[2 1 3 4 5 6 9 8 10 7]
```

```
sigma =
```

2	1	3	4	5	6	9	8	10	7
---	---	---	---	---	---	---	---	----	---

```
>> ApplyPermutation (GB,sigma,n)
```

```
ans =
```

0	1	0	1	1	0	0	0	1	1
1	0	1	1	0	1	0	1	0	0
0	1	0	0	0	1	1	1	1	0
1	1	0	0	1	1	0	0	1	0
1	0	0	1	0	0	1	1	0	1
0	1	1	1	0	0	1	0	0	1
0	0	1	0	1	1	0	1	1	0
0	1	1	0	1	0	1	0	0	1
1	0	1	1	0	0	1	0	0	1
1	0	0	0	1	1	0	1	1	0

Using the resulting graph N10E5', we proceeded to the test as follows:

```
eestudent@eeuser-LINUX:~/Desktop/nauty22$ ./dreadnaut
Dreadnaut version 2.2 (32 bits).
> n=10 g
 0 : 1 2 3 5 7;
 1 : 3 4 6 8;
 2 : 5 6 7 9;
 3 : 4 5 6;
 4 : 7 8 9;
 5 : 8 9;
 6 : 8 9;
 7 : 8 9.
> x @
10 orbits; grpsize=1; 0 gens; 11 nodes (7 bad leaves); maxlev=2
tctotal=10; canupdates=3; cpu time = 0.00 seconds
> g
 0 : 1 3 4 8 9;
 1 : 2 3 5 7;
 2 : 5 6 7 8;
 3 : 4 5 8;
 4 : 6 7 9;
 5 : 6 9;
 6 : 7 8;
 7 : 9;
 8 : 9.
> x
10 orbits; grpsize=1; 0 gens; 11 nodes (8 bad leaves); maxlev=2
tctotal=10; canupdates=2; cpu time = 0.00 seconds
> ##
h and h' are identical.
 0-1 1-0 2-2 3-3 4-4 5-5 6-8 7-7 8-9 9-6
```

Nauty has in this case also solved the problem in less than a second. It also provides the permutation which applied to N10E5 gives us N10E5'.

Test8: Graph-N14E7

```
>> n=14
```

```
n =
```

```
14
```

```
>> GB=[0 1 1 1 0 1 0 1 0 0 1 0 1 0;1 0 0 1 1 0 1 0 1 0 0 1 1 0;1 0 0 0
0 1 1 1 0 1 1 0 0 1;1 1 0 0 1 1 1 0 0 0 0 1 1 0;0 1 0 1 0 0 0 1 1 1 1 0
0 1;1 0 1 1 0 0 0 0 1 1 1 0 1 0;0 1 1 1 0 0 0 0 1 1 0 1 0 1;1 0 1 0 1 0
0 0 1 1 1 0 1 0;0 1 0 0 1 1 1 1 0 0 0 1 1 0; 0 0 1 0 1 1 1 1 0 0 0 1 0
1;1 0 1 0 1 1 0 1 0 0 0 1 0 1;0 1 0 1 0 0 1 0 1 1 1 0 0 1;
1 1 0 1 0 1 0 1 1 0 0 0 0 1;0 0 1 0 1 0 1 0 0 1 1 1 1 0]
```

```
GB =
```

```
0 1 1 1 0 1 0 1 0 0 1 0 1 0
1 0 0 1 1 0 1 0 1 0 0 1 1 0
1 0 0 0 0 1 1 1 0 1 1 0 0 1
1 1 0 0 1 1 1 0 0 0 0 1 1 0
0 1 0 1 0 0 0 1 1 1 1 0 0 1
1 0 1 1 0 0 0 0 1 1 1 0 1 0
0 1 1 1 0 0 0 0 1 1 0 1 0 1
1 0 1 0 1 0 0 0 1 1 1 0 1 0
0 1 0 0 1 1 1 1 0 0 0 1 1 0
0 0 1 0 1 1 1 1 0 0 0 1 0 1
1 0 1 0 1 1 0 1 0 0 0 1 0 1
0 1 0 1 0 0 1 0 1 1 1 0 0 1
1 1 0 1 0 1 0 1 1 0 0 0 0 1
0 0 1 0 1 0 1 1 1 1 0
```

```
>> sigma=[1 5 2 4 7 8 3 6 10 9 11 13 12 14]
```

```
sigma =
```

```
1 5 2 4 7 8 3 6 10 9 11
13 12 14
```

```
>> ApplyPermutation (GB,sigma,n)
```

```
ans =
```

```
0 1 0 1 1 1 0 1 0 0 1 1 0 0
1 0 1 0 0 1 0 1 1 0 1 0 0 1
0 1 0 1 1 0 0 0 1 1 0 0 1 1
1 0 1 0 1 0 1 1 0 0 0 1 1 0
1 0 1 1 0 0 1 0 0 1 0 1 1 0
1 1 0 0 0 0 1 0 1 1 1 1 0 0
0 0 0 1 1 1 0 0 1 1 1 0 0 1
1 1 0 1 0 0 0 0 1 1 1 1 0 0
0 1 1 0 0 1 1 1 0 0 0 1 1
0 0 1 0 1 1 1 1 0 0 0 1 1 0
```

1	1	0	0	0	1	1	1	0	0	0	0	1	1
1	0	0	1	1	1	0	1	0	1	0	0	0	1
0	0	1	1	1	0	0	0	1	1	1	0	0	1
0	1	1	0	0	0	1	0	1	0	1	1	1	0

The results are as follows:

```
eestudent@eeuser-LINUX:~/Desktop/nauty22$ ./dreadnaut
Dreadnaut version 2.2 (32 bits).
> n=14 g
0 : 1 2 3 5 7 10 12;
1 : 3 4 6 8 11 12;
2 : 5 6 7 9 10 13;
3 : 4 5 6 11 12;
4 : 7 8 9 10 13;
5 : 8 9 10 12;
6 : 8 9 11 13;
7 : 8 9 10 12;
8 : 11 12;
9 : 11 13;
10 : 11 13;
11 : 13;
12 : 13.
> c -a -m
> x@
14 orbits; grpsize=1; 0 gens; 15 nodes (12 bad leaves); maxlev=2
tctotal=14; canupdates=2; cpu time = 0.00 seconds
> g
0 : 1 3 4 5 7 10 11;
1 : 2 5 7 8 10 13;
2 : 3 4 8 9 12 13;
3 : 4 6 7 11 12;
4 : 6 9 11 12;
5 : 6 8 9 10 11;
6 : 8 9 10 13;
7 : 8 9 10 11;
8 : 12 13;
9 : 11 12;
10 : 12 13;
11 : 13;
12 : 13.
> x
14 orbits; grpsize=1; 0 gens; 15 nodes (12 bad leaves); maxlev=2
tctotal=14; canupdates=2; cpu time = 0.00 seconds
> ##
h and h' are identical.
0-0 1-4 2-1 3-3 4-6 5-7 6-2 7-5 8-9 9-8 10-10 11-12 12-11 13-13
```

We also tested non-isomorphic graphs with Nauty.

Test9: Graph-N10E5

We created N10E5' from N10E5 such that the two graphs are not isomorphic.

```
eestudent@eeuser-LINUX:~/Desktop/nauty22$ ./dreadnaut
Dreadnaut version 2.2 (32 bits).
> n=10 g
 0 : 1 2 3 5 7;
 1 : 3 4 6 8;
 2 : 5 6 7 9;
 3 : 4 5 6;
 4 : 7 8 9;
 5 : 8 9;
 6 : 8 9;
 7 : 8 9.
> c-a-m
> x @
10 orbits; grpsize=1; 0 gens; 11 nodes (7 bad leaves); maxlev=2
tctotal=10; canupdates=3; cpu time = 0.00 seconds
> g
 0 : 1 3 4 8 9
 0 : 1 3 4 8 9;
 1 : 2 3 5 6;
 2 : 5 6 7 8;
 3 : 4 5 8;
 4 : 6 7 9;
 5 : 6 9;
 6 : 7 8;
 7 : 9;
 8 : 9.
> x
10 orbits; grpsize=1; 0 gens; 1 node; maxlev=1
tctotal=0; canupdates=1; cpu time = 0.00 seconds
> ##
h and h' are different.
```

In this case also, Nauty found the answer in less than a second.

6. CONCLUSIONS

Our experimental results provide partial conclusions only at this point. We intend to continue this work during the next several months to obtain conclusive results regarding the suitability of graph isomorphism based approach for ZKP protocol implementation. We are experimenting with the following types of graphs.

1. Regular, but not strongly regular graphs with the same number of connections (degree) for each node,
2. Graphs with low number of automorphs, and
3. Node degree about $1/2$ of the number of nodes

We plan to generate graphs with the above characteristics and analyze the complexity of breaking their graph isomorphism using Nauty and other tools such as VF2.

7. REFERENCES

- [1] R. Norville, K. R. Namuduri, and R. Pendse, “Node Authentication in Networks using Zero-Knowledge Proofs”, in “Web Services Security and E-Business”, pp. 143-164, Idea Group Publishing, 2007.
- [2] B. D. McKay, “Practical Graph Isomorphism”, *Congressus Numerantium*, Vol. 30, pp. 45-87, 1981.
- [3] B. D. McKay, “*nauty* User’s Guide (Version2.2)”.

8. APPENDIX A

ZKP PROTOCOL

a) *Main Code*

```
i=0;j=0;
n = input('The value of n is:');
Repeat = input ('Start the interaction? (1: Yes, 0: No):');

G1 = input('The value of the original graph G1 is: G1=');

pi = input('The value of the secret pi is:');

ro = input('The value of ro is:');

%Creating the graph G2
G2=MakeIsomorphic(G1,pi,n);

while Repeat == 1
    g = input('Choose the graph from which to create the graph H to
give the verifier (0:G1,1:G2):');

    if g==0
        H=MakeIsomorphic(G1,ro,n);
    else
        H=MakeIsomorphic(G2,ro,n);
    end

    q = input ('The verifier randomly chooses a graph for the prover
to generate(1:G1,2:G2):');

    sigma = input('The value of sigma is:');

    if(q==0)

        R=ApplyPermutation(H,sigma,n);
        CheckGraphs(G1,R,n);

    elseif(q==1)

        R=ApplyPermutation(H,sigma,n);
        CheckGraphs(G2,R,n);
    end

    Repeat=input('\nDoes the verifier want to repeat the
process?(1:Yes,0:No):');

End
```

b) “MakeIsomorphic” function

```
function[H] = MakeIsomorphic(Ga,A,n)
for i=1:n
    for j=1:n
        H(A(i),A(j))= Ga(i,j);
    end
end
```

c) “ApplyPermutation” function

```
function [R]= ApplyPermutation (GB,sigma,n)
for i=1:n
    for j=1:n
        R(sigma(i),sigma(j))=GB(i,j);
    end
end
```

d) “CheckGraphs” function

```
function[x]= CheckGraphs(GB,R,n)
p=1;
for i=1:n
    for j=1:n
        if GB(i,j)~=R(i,j)
            p=0;
            break;
        end
    end
end

if p==0
    disp ('The prover is proved wrong! He does not know the secret!');
elseif p==1
    disp('The prover has proved himself!');
end
```

e) “*CreateSigma*” function

```
function[sigma] = CreateSigma (a,b,pi,ro,n)
for i=1:n
if (a==0&&b==0 || a==1&&b==1)

    sigma(ro(i))=i;

elseif(a==1&&b==0)

    sigma(ro(pi(i)))=i;

elseif(a==0&&b==1)

    sigma(ro(i))= pi(i);
end
end
end
```

9. APPENDIX B

ABSTRACT OF THE PRESENTATION MADE IN NATO SPONSORED CONFERENCE HELD IN SLOVENIA, PRESENTED BY BRENT HOLMES, KAMESH NAMUDURI AND CAPT. RICO CODY.

Trust is a fundamental concept that enables cooperation and collaboration among the nodes in any network. Formal trust models are necessary for sharing information in a collaborative environment. Trust assessment methods that are commonly used in terrestrial network applications or in social networks passively gather information about other nodes and take significant amount of time for assessing trust. Such models are not suitable for tactical airborne networks which are typically deployed for short durations of time.

This paper presents an active trust model in which the nodes in a network proactively probe other nodes to assess their level of trust before sharing mission specific information. Active trust models are useful for assessing trust within short durations of time making them appropriate for airborne networks. The proposed model is based on zero-knowledge proofs.

10. APPENDIX C

ABSTRACT OF THE PRESENTATION MADE IN INTERNATIONAL CONFERENCE ON COMPUTING, COMMUNICATIONS, AND NETWORKING HELD IN KARUR, INDIA

Title: Consensus building in cooperative decentralized systems

This talk discusses approaches to consensus building and task assignment problems in decentralized systems consisting groups of collaborating nodes. Consensus building is an important and fundamental problem in numerous real-world applications including disaster recovery, and search and rescue operations conducted by teams of people. Exploration of planetary surfaces by teams of robots and aerial surveillance for ground targets are some of the high-end applications that require elegant theoretical models and solutions to task assignment.

The problem setting for the research issues addressed in this talk is as follows. Consider a decentralized system (say, an ad hoc network) consisting of N number of nodes, divided into M groups, with each group formed by a variable number of nodes. In each group, one node serves as a leader and functions as a fusion center. All nodes in the system are assumed to have the capability to communicate with their group members. A node may participate as a member in more than one group. All nodes are capable of sensing, processing information locally, and communicating with their leaders. Direct communication among peers is not allowed, i.e., member nodes are not allowed to communicate with each other, and leaders are not allowed to communicate with other leaders. Each node processes the information it gathers about the phenomenon that it observes, and communicates its observation to the leader that it is connected to. The objective is to design strategies for decision making and task scheduling, which are central to such decentralized cooperative systems. Decision making involves coming to an agreement on the observed phenomenon. Task scheduling involves coming to consensus on the one-to-one mapping between the nodes and the tasks to be accomplished. The proposed strategies are based on two fundamental ideas originated from the two different fields: (1) decision fusion based on belief propagation and self-organization based on coupled selection equations in dynamic control systems. Decision fusion based on belief propagation is relatively a new area of research in decentralized systems. Developed by Gallager with applications in channel coding and communications in 1963, belief propagation makes use of local interactions to arrive at a global consensus. It is applicable for ad hoc networks such as airborne networks, sensor networks, and social networks in which the nodes are typically connected with only their neighboring nodes.

11. APPENDIX D

ABSTRACT OF THE PAPER SUBMITTED TO NSF SPONSORED MEETING ON CYBER PHYSICAL SYSTEMS (WITH BRENT HOLMES)

The aviation infrastructure forms an important component of the nation's critical infrastructure. By considering the issues involved in protecting the onboard aviation information in conjunction with the introduction of Internet access within an airplane, we propose to investigate the risks, vulnerabilities, and threats in providing network connectivity to airplanes and a comprehensive set of supplementary security mechanisms to address these issues.

The first objective of the proposed discussion is to develop security and fault-tolerance solutions to the existing air-ground communication systems. The second objective is to investigate, design, and develop off-board control methods that would enable the ground crew to override the on-board control mechanism in the event of malicious takeover of an airplane. These objectives will be achieved by investigating the following four specific aspects of onboard information assurance.

- (1) Identifying the risks, vulnerabilities, and threats
- (2) Methods for downloading onboard aircraft parameters
- (3) Air-ground network security and performance, and
- (4) Off-line control mechanisms

The most significant result of the proposed work is a set of guidelines and best practices for securing onboard aircraft information suitable for practical implementation. The proposed activities also open up avenues for greater collaboration between universities and aviation industry resulting in safety and security of our nation's aviation infrastructure.